Explicit Model Checking of Very Large MDP using Partitioning and Secondary Storage*

Arnd Hartmanns and Holger Hermanns

Saarland University – Computer Science, Saarbrücken, Germany

Abstract The applicability of model checking is hindered by the state space explosion problem in combination with limited amounts of main memory. To extend its reach, the large available capacities of secondary storage such as hard disks can be exploited. Due to the specific performance characteristics of secondary storage technologies, specialised algorithms are required. In this paper, we present a technique to use secondary storage for probabilistic model checking of Markov decision processes. It combines state space exploration based on partitioning with a block-iterative variant of value iteration over the same partitions for the analysis of probabilistic reachability and expected-reward properties. A sparse matrix-like representation is used to store partitions on secondary storage in a compact format. All file accesses are sequential, and compression can be used without affecting runtime. The technique has been implemented within the Modest Toolset. We evaluate its performance on several benchmark models of up to 3.5 billion states. In the analysis of time-bounded properties on real-time models, our method neutralises the state space explosion induced by the time bound in its entirety.

1 Introduction

Model checking [9] is a formal verification technique to ensure that a given model of the states and behaviours of a safety- or performance-critical system satisfies a set of requirements. We are interested in models that consider *nondeterminism* as well as quantitative aspects of systems in terms of *time* and *probabilities*. Such models can be represented as Markov decision processes (MDP [32]) and verified with *probabilistic model checking*. However, the applicability of model checking is limited by the state space explosion problem: The number of states of a model grows exponentially in the number of variables and parallel components, yet they have to be represented in limited computer memory in some form. Probabilistic model checking is particularly affected due to its additional numerical complexity. Several techniques are available to stretch its limits: For example, symbolic probabilistic model checking [2], implemented in the Prism tool [25], uses variants of binary decision diagrams (BDD) to compactly represent the state spaces

* This work is supported by the EU 7th Framework Programme under grant agreements 295261 (MEALS) and 318490 (SENSATION), by the DFG as part of SFB/TR 14 AVACS, by the CAS/SAFEA International Partnership Program for Creative Research Teams, and by the CDZ project CAP (GZ 1023).

of well-structured models in memory at the cost of verification runtime. Partial order [4] and confluence reduction [35] deliver smaller-but-equivalent state spaces and work particularly well for highly symmetric models. When trading accuracy for tractability or efficiency is acceptable, abstraction and refinement techniques like CEGAR [23] can be applied. The common theme is that these approaches aim at reducing the state space or its representation such that it fits, in its entirety, into the main memory of the machine used for model checking. An alternative is to store this data on secondary storage such as hard disks or solid state drives and only load small parts of it into main memory when and as needed. This is attractive due to the vast difference in size between main memory and secondary storage: Typical workstations today possess in the order of 4-8 GB of main memory, but easily 1 TB or more of hard disk space. Moreover, with the advent of dynamically scalable cloud storage, virtually unlimited off-site secondary storage has become easily accessible. For conciseness, we from now on refer to main memory as *memory* and to any kind of secondary storage as *disk*.

In this paper, we present a method and tool implementation for disk-based probabilistic model checking of MDP. Any such approach must solve two tasks: State space *exploration*, the generation and storage on disk of a representation of the reachable part of the state space, and the disk-based *analysis* to verify the given properties of interest based on this representation. The core challenge is that the most common type of secondary storage, magnetic hard disks, exhibits extremely low random-access performance, yet standard memory-based methods for exploration and analysis access the state space in a practically random way.

Previous work. Exploration is an implicit graph search problem, and a number of solutions that reduce the amount of random accesses during search have been proposed in the literature. These fall into three broad categories: (*i*) exploiting the layered structure of breadth-first search (BFS) by keeping only the current BFS layer in memory while delaying duplicate detection w.r.t. previous layers until the current one has been fully explored [12,33]; (*ii*) partitioning the state space according to some given or automatically computed partitioning function over the states and then loading only one partition into memory at a time in an iterative process [5,16]; (*iii*) treating memory purely as a cache for a disk-based search, but using clever hashing and hash partitioning techniques to reduce and sequentialise disk accesses [19]. Exploration can naturally be combined on-the-fly with checking for the reachability of error states, and methods to perform on-the-fly verification of liveness and LTL properties exist [6,13,15].

The analysis of other logics, such as CTL model checking with satisfaction sets, and of other models, such as probabilistic model checking of MDP with value iteration, inherently require the entire state space for a dedicated analysis step following exploration. Previous work on disk-based probabilistic model checking considers purely stochastic models and focusses on the analysis phase: In absence of nondeterminism, classical block-iterative methods [34] can be used with disk-based (sparse) matrix representations of Markov models. They proceed by loading into memory and analysing one matrix block at a time (plus those that

it depends on) iteratively until the method has converged for all blocks. Implementations can be divided into *matrix-out-of-core* and *complete out-of-core* approaches [30]. In the former, the vector of state values being iteratively computed is still kept in memory in its entirety [11]. It is similar to how PRISM [25] uses BDD in its "HYBRID" engine for the model only, while both model and values are represented symbolically in its "MTBDD" engine. The symbolic and disk-based approaches for Markov chains can be combined [24]. Further work on the disk-based analysis of purely stochastic models includes different implementations that are both disk-based and parallelised or distributed [7,20].

For the nondeterministic-probabilistic model of MDP that we are concerned with, the default scalable analysis algorithm used in model checking is value iteration, an iterative fixpoint method that updates the values of each state based on a function over the values of its immediate successors until all changes remain below a given error. We are aware of only one explicitly disk-based approach to value iteration, which associates the values to the transitions instead of the states and is based on sequentially traversing two files containing the transitions that have been externally sorted by source and target states in each iteration [14]. However, external sorting is a costly operation, leading to high runtime.

The correctness of value iteration depends neither on the order in which the updates are performed nor on how many updates a state receives in one iteration. This can be exploited to improve its performance by taking the graph structure of the underlying model into account to perform more updates for "relevant" states in a "good" order. One such technique is topological value iteration [10], based on a division of the MDP into strongly connected components. More generally, this means that value iteration can also be performed in a block-iterative manner.

Our contribution. The technique for disk-based probabilistic model checking of MDP that we present in this paper is a complete out-of-core method. It combines the state space partitioning approach from disk-based search with a block-iterative variant of value iteration based on a very compact sparse matrix-like representation of the partitions on disk. In light of the disk space available, compactness seems at first sight to be a non-issue, but in fact is a crucial aspect due to the low throughput of hard disks compared to main memory. Based on a given partitioning function, our approach proceeds by first exploring the partitions of the state space using an explicit state representation while directly streaming the sparse matrix-like representation to disk. When exploration is completed, the stored partitions are analysed using a block-based variant of value iteration: It iterates in an outer loop over the partitions on disk, for each of which value iterations are performed in an inner loop until convergence. All read and write operations on the files we generate on disk are sequential. We can thus easily add compression, which in our experiments reduces the amount of disk space needed by a factor of up to 10 without affecting overall runtime.

Our method has been implemented by extending the mcsta tool [18] of the MODEST TOOLSET [22]. The implementation currently supports the computation of reachability probabilities and expected accumulated rewards. To the best

of our knowledge, mcsta is at this point the only publicly available tool that provides disk-based verification of MDP. We have evaluated the approach and its implementation on five case studies. The largest model we consider has 3.5 billion states. It can be explored and analysed in less than 8 hours using no more than 2 GB of memory and 30 GB disk space. Our technique is particularly efficient for the analysis of time-bounded properties on real-time extensions of MDP. In these cases, the overhead of using the disk is small and the enormous state space explosion caused by the time bounds can be neutralised in its entirety.

2 Preliminaries

The central formal model that we use are Markov decision processes:

Definition 1. *A* probability distribution *over a countable set Ω is a function $\mu \in \Omega \to [0,1]$ such that $\sum_{\omega \in \Omega} \mu(\omega) = 1$. Its* support *is* $\mathrm{support}(\mu) = \{\, s \in S \mid \mu(s) > 0 \,\}$. *We denote by* $\mathrm{Dist}(\Omega)$ *the set of all probability distributions over Ω.*

Definition 2. *A* Markov decision process *(MDP) is a triple $\langle S, T, s_0 \rangle$ consisting of a countable set of* states *S, a* transition function *$T \in S \to 2^{\mathrm{Dist}(S \times R)}$ for a countable subset $R \subsetneq \mathbb{R}$ with $T(s)$ countable for all $s \in S$, and an* initial state *$s_0 \in S$. A* partitioning function *for an MDP is a function $f \in S \to \{\, 1, \ldots, k \,\}$ for some $k \in \mathbb{N}$ with $f(s_0) = 1$.*

For $s \in S$, we call $\mu \in T(s)$ a *transition* of s, and a pair $b = \langle s', r \rangle \in \mathrm{support}(\mu)$ a *branch* of μ, with s' being the *target state* of b and r being the associated *reward* value. MDP support both nondeterministic and probabilistic choices: A state can have multiple outgoing transitions, each of which leads into a probability distribution over pairs $\langle s, r \rangle$. A partitioning function $f \in S \to \{\, 1, \ldots, n \,\}$, $n \in \mathbb{N}$, divides the states of an MDP into partitions $P_i = \{\, s \in S \mid f(s) = i \,\}$. The *partition graph* is the directed graph $\langle P, U \rangle$ with nodes $P = \{\, P_i \mid 1 \le i \le k \,\}$ and edges $U = \{\, \langle P_i, P_j \rangle \mid i \neq j \wedge \exists\, s \in P_i, \mu \in T(s), \langle s', r \rangle \in \mathrm{support}(\mu) \colon s' \in P_j \,\}$. It is *forward-acyclic* if there is no $\langle P_i, P_j \rangle \in U$ with $j < i$.

We are interested in the probability of reaching certain states in an MDP and in the expected reward accumulated when doing so. Since an MDP may contain nondeterministic choices, these values are only well-defined under a *scheduler*, which provides a recipe to resolve the nondeterminism. The verification questions are thus: Given a set of states $F \subseteq S$, (*i*) what is the maximum/minimum probability of eventually reaching a state in F over all possible schedulers (*reachability probability*), and (*ii*) what is the maximum/minimum expected accumulated reward once a state in F is reached for the first time over all possible schedulers (*expected reward*)? These quantities can be formally defined using the usual cylinder set construction for the paths of the MDP [17].

The computation of these quantities is typically done using *value iteration*, as shown in Algorithm 1 for maximum reachability probabilities. For the minimum case, we replace maximisation by minimisation in line 5. To compute expected rewards, a precomputation step is needed to determine those states from which

```
1 values := { s ↦ 1 | s ∈ F } ∪ { s ↦ 0 | s ∈ S \ F }          // the value vector
2 repeat
3 |   error := 0
4 |   foreach s ∈ S \ F do
5 |   |   v_new := max { Σ_{⟨s',r⟩∈support(μ)} μ(s') · values(s') | μ ∈ T(s) }
6 |   |   if v_new > 0 then error := max{ error, |v_new − values(s)|/values(s) }
7 |   |   values(s) := v_new
8 until error < ε
9 return values(s_0)
```

Algorithm 1: Value iteration to compute max. reachability probabilities

F is reachable with probability one and zero, respectively. This can be done with straightforward fixpoint algorithms over the graph structure of the MDP [17].

Using MDP directly to build models of complex systems is cumbersome. Instead, higher-level formalisms such as PRISM's guarded command language are used. They add to MDP variables that take values from finite domains. In an *MDP with variables* (VMDP), each transition is associated with a *guard*, a Boolean expression that disables the transition when it is *false*. The probabilities and reward values of the branches are given as real-valued arithmetic expressions. Every branch has an *update* that assigns new values (given as expressions) to the variables of the process. The semantics of a VMDP M is the MDP $[\![M]\!]$ whose states are pairs $\langle s, v\rangle$ of a state s of M and a valuation v for the variables. Transitions out of s that are disabled according to v do not appear in $[\![M]\!]$, and the valuations of a branch's targets are computed by applying the update of the branch to the valuation of the transition's source state. A partitioning function f for a VMDP can be determined by an upper-bounded arithmetic expression e with values in $\mathbb{N}$: $f(\langle s, v\rangle) = e(v)$ where $e(v)$ is the evaluation of e in v. The reachability set F can likewise be characterised by a Boolean expression.

Real-time extensions of MDP To model and analyse real-time systems, MDP can be extended with real-valued clock variables and state invariant expressions as in timed automata (TA [3]), leading to the model of probabilistic timed automata (PTA [27]). A number of techniques are available to model-check PTA [31], but only the digital clocks approach [26] allows the computation of both reachability probabilities and expected rewards: Clocks are replaced by bounded integer variables, and self-loop transitions are added to increment them synchronously as long as the state invariant is satisfied. This turns the (finite) PTA into a (finite) VMDP. The conversion preserves reachability probabilities and expected reward values whenever all clock constraints in the PTA are closed and diagonal-free. However, the size of the final MDP is exponential in the number of clock variables and the maximum constants that they are compared to.

For timed models, we are also interested in *time-bounded reachability*: Ranging over all possible schedulers, what is the maximum/minimum probability of

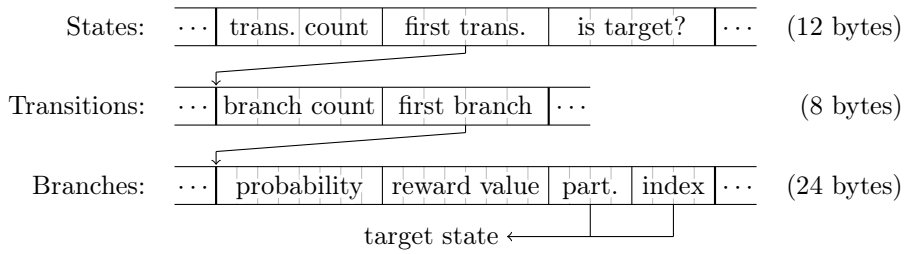

Figure 1. In-memory representation of MDP for fast random access

eventually reaching a state in F within at most t time units? These probabilities can be computed by adding a new clock variable x to the PTA that is never reset and computing the reachability probability for the set $F' = \{ \langle s, v \rangle \mid s \in F \wedge v(x) \leq t \}$ in the resulting digital clocks MDP [31].

A further extension of PTA are stochastic timed automata (STA [8]). They allow assignments of the form $x := \text{SAMPLE}(D)$ to sample from (continuous) probability distributions D, e.g. exponential or normal distributions, in updates. This allows for stochastic delays, such as the exponentially-distributed sojourn times of continuous-time Markov chains, in addition to the nondeterministic delays of (P)TA. A first model checking technique for STA has recently been described [18] and implemented within the mcsta tool of the MODEST TOOLSET [22]. It works by abstracting assignments that use continuous distributions into finite-support probabilistic choices plus continuous nondeterminism, turning the STA into a PTA that can be analysed with e.g. the digital clocks technique.

3 Disk-Based State Space Exploration with Partitioning

In this section, we describe the partitioned state space exploration approach that we use in our disk-based analysis technique for MDP. We assume that the MDP to be explored is given in some compact description that can be interpreted as a VMDP, and a partitioning function f is given as an expression over its variables. Disk-based exploration using partitioning has been the subject of previous work [5,16], so we focus on the novel aspect of generating a sparse matrix-like representation of the MDP on-the-fly during explicit-state exploration with low memory usage and in a compact format in a single file on disk.

3.1 Representation of MDP in Memory and on Disk

There are conceptually two ways to represent in memory an MDP that is the semantics of a VMDP: In an *explicit-state* manner, or in a *sparse matrix-style* representation. In the former, only the set of states of the MDP is kept, with each state stored as a vector $\langle s, v = \langle v_1, \ldots, v_n \rangle \rangle$ where s identifies the state in the original VMDP and v_i the value of its i-th variable. Given a state and the compact description of the VMDP, we can recompute transitions and branches

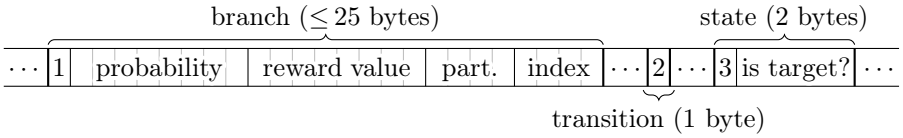

Figure 2. Inverse-sequential format to compactly represent MDP on disk

at any time on-demand. The other alternative is to identify each of the n states of the MDP with a value in $\{1, \ldots, n\}$, its *index*, and explicitly store the set of transitions belonging to a state index and the transitions' branches. For each branch, its probability, its reward value, and the index of the target state need to be stored. This sparse matrix-style representation takes its name from the similar idea of storing a Markov chain as a sparse encoding of its probability matrix. All information about the inner structure of the states is discarded.

Figure 1 outlines the sparse matrix-style representation used by mcsta, which keeps three arrays to store the states, transitions and branches of a partition of the state space. For a state, "is target?" is *true* iff it is in the reachability set F that we consider. The target state of a branch is identified by its partition and its relative index within that partition. This format is more memory-efficient than an explicit-state representation when the model has many variables, and access to transitions and branches can be significantly faster because guards and other expressions in the model do not need to be evaluated on every access.

The format of Figure 1 allows fast random access to all parts of the state space. However, when only sequential access is required, an MDP can be stored more compactly. Figure 2 shows the "inverse-sequential" format used by our technique to store state spaces on disk. States, transitions and branches are stored as a sequence of records, with the type of each record given by its first byte. Branches can be stored even more compactly by adding record types for common cases such as branches with probability 1. The key idea of the format is to first store all the branches of a transition before the transition record itself, and similarly store all the transitions (each preceded by its branches) of a state before the state record itself. In this way, we do not need to store the number of transitions and the index of the first transition for a state since its transitions are precisely those that appeared since the previous state record (and analogously for the branches of a transition). The random-access format of Figure 1 can be reconstructed from a single sequential read of a file in the inverse-sequential format, and the file can be created sequentially with one simultaneous sequential pass through the arrays of the random-access format in memory.

3.2 Disk-Based Exploration using Partitioning

Our disk-based exploration technique is given as Algorithm 2. It is based on the approach of [5,16]. Files on disk are indicated by subscript $_D$; when loaded into memory, the corresponding variable has subscript $_M$. For each partition, we use BFS to discover new states (lines 12 to 38) with the following data in memory:

```
1  int count := 1, queue^1_D.append(s_0)
2  repeat
3      changed := false
       // iterate over all partitions discovered so far
4      for i := 1 to count do
           // Phase 1: update preliminary target indices for cross transitions
5          foreach j ∈ successors^i do array updates^j_M := updates^j_D.load()
6          oldmatrix^i_D := matrix^i_D, matrix^i_D.clear()                    // rename file
7          foreach r ∈ oldmatrix^i_D do                                       // read records sequentially
8              if r = ⟨1, p, r, j, k⟩ ∧ k < 0 then
9                  matrix^i_D.append(⟨1, p, r, j, updates^j_M[−k − 1]⟩)       // update index
10             else matrix^i_D.append(r)
11         unload updates^j_M for all j ∈ successors^i
           // Phase 2: explore more states in breadth-first manner
12         updates^i_D.clear()
13         queue queue^i_M := queue^i_D.load(), queue^i_D.clear(), qlen^i := 0
14         indexed-set states^i_M := states^i_D.load()
15         set done^i := states^i_M
16         while queue^i_M.length > 0 do
17             explicit-state s := queue^i_M.dequeue()
18             if s ∉ states^i_M then states^i_M.add(s), states^i_D.append(s)
19             updates^i_D.append(states^i_M.indexof(s))
20             if s ∈ done^i then continue else changed := true
21             foreach t ∈ s.transitions() do
22                 if ¬t.guard(s.v) then continue
23                 foreach b ∈ t.branches() do
24                     double p := b.probability(s.v), r := b.reward(s.v)
25                     if p = 0 then continue
26                     explicit-state s′ := b.target(s.v)
27                     if f(s′) = i then                                      // local transition
28                         if s′ ∉ states^i_M then states^i_M.add(s′), states^i_D.append(s′)
29                         queue^i_M.enqueue(s′)
30                         matrix^i_D.append(⟨1, p, r, i, states^i_M.indexof(s′)⟩)
31                     else                                                   // cross transition
32                         j := f(s′), successors^i.add(j), count := max{ count, j }
33                         queue^j_D.append(s′), qlen^j = qlen^j + 1
34                         matrix^j_D.append(⟨1, p, r, j, −qlen^j⟩)  // prelim. index < 0
35                 matrix^i_D.append(⟨2⟩)
36             matrix^i_D.append(⟨3, s ∈ F⟩)
37             done^i.add(s)
38         unload queue^i_M, states^i_M, done^i
39 while changed
```

Algorithm 2: Partitioned disk-based exploration with sparse matrix creation

- $states^i$: The set of states (explicit-state representation) of partition i is loaded into memory in its entirety when search begins for the partition (line 14). States are added in memory and appended on disk (lines 18 and 28).
- $queue^i$: The queue of states to explore in partition i. When a cross-transition is found during search in partition i, i.e. a branch leads to another partition $j \neq i$, then the target state is appended to $queue^j_{\mathrm{D}}$ on disk (line 33). For local transitions, the target state is appended to $queue^i_{\mathrm{M}}$ in memory (line 29).
- $done^i$: The in-memory set of fully explored states for the current iteration.

When an iteration of search in partition i ends, $states^i$ is backed on disk, $queue^i$ is empty, and $done^i$ is no longer needed, so we remove them from memory (line 38).

During search, we simultaneously create the sparse matrix-like representation of the partitions on disk in files $matrix^i_{\mathrm{D}}$ using the inverse-sequential format. The files are not loaded into memory. The records for new branches, transitions and states are appended to the file in lines 30, 34, 35 and 36. The main complication is the correct treatment of cross transitions: A branch record stores the partition j of its target state s' and the index of s' within that partition. However, we cannot determine this index without loading all of $states^j_{\mathrm{D}}$ into memory, and even then, s' may not have been explored yet. To solve this problem, we instead use the index of s' in $queue^j_{\mathrm{D}}$, which is easily determined (line 33). To distinguish such a preliminary index, which needs to be corrected later, from a local or already corrected one, we store it as a negative value (line 34).

The correction of these preliminary indices inside $matrix^i_{\mathrm{D}}$ happens at the beginning of an iteration for partition i (lines 5 to 11). The files $updates^j_{\mathrm{D}}$ for all successor partitions j are loaded into memory. These files have been created by the previous iteration for partition j in lines 12 and 19 and contain the correct indices for all states that were previously in $queue^j_{\mathrm{D}}$, at the same position. The preliminary queue-based indices in partition i can thus be corrected by a sequential pass through its sparse matrix-like representation in file $matrix^i_{\mathrm{D}}$, replacing all negative indices $-k$ for partition j by the corrected value at $updates^j_{\mathrm{M}}[k]$. This is a random-access operation on the files $updates^j_{\mathrm{D}}$, which is why they were loaded into memory beforehand, but a sequential operation on the file $matrix^i_{\mathrm{D}}$, of which we thus only need to load into memory one record at a time. Observe that this correction process relies on the availability of $updates^j_{\mathrm{D}}$ for all successor partitions j. To assure this, we iterate over all partitions in a fixed order in line 4 instead of always moving to the partition with the longest queue as in [5,16].

To describe the memory usage and I/O complexity of this algorithm, let $n_{\max}$ denote the max. number of states, $s_{\max}$ the max. number of successor partitions (i.e. the max. outdegree of the partition graph), and $c_{\max}$ the max. number of *incoming* cross edges, over all partitions. Then the correction of preliminary indices in phase 1 needs memory in $O(s_{\max} \cdot c_{\max})$ for the $updates^j_{\mathrm{M}}$ arrays and the exploration in phase 2 needs memory in $O(n_{\max} + c_{\max})$ for $states^i_{\mathrm{M}}$ and $done^i$ plus $queue^i_{\mathrm{M}}$. Additionally, we need memory for the sets of integers $successors^i$, which we assume to be negligible compared to the other data items. A theoretical analysis of the I/O complexity [1] of a partitioning-based technique is problematic (and in fact absent from [5] and [16]) due to the way multiple files

```
1  for i := 1 to count do                                   // prepare value arrays on disk
2      matrix^i_M := matrix^i_D.load()
3      for k := 0 to matrix^i_M.states.length − 1 do
4          values^i_D.append(matrix^i_M.states[k].istarget ? 1 : 0)
5      unload matrix^i_M
6  while changed do                                         // block-iterative value iteration
7      changed := false                                     // changed is initially false
8      for i := count down to 1 do
9          matrix^i_M := matrix^i_D.load(), values^i_M := values^i_D.load()
10         foreach j ∈ successors^i do values^j_M := values^j_D.load()
11         repeat
12             error := 0
13             for k := 0 to matrix^i_M.states.length − 1 do
14                 if matrix^i_M.states[k].istarget then continue
15                 v_new := max ... // as in Algorithm 1, but with values^i_M/values^j_M
16                 if v_new > 0 then error := ...// compute error as in Algorithm 1
17                 values^i_M[k] := v_new
18                 if error ≥ ε then changed := true
19         until error < ε
20         unload matrix^i_M, values^i_M and the values^j_M for all j ∈ successors^i
21 return values^1_D[0]
```

Algorithm 3: Partitioned value iteration for max. reachability probabilities

are used e.g. when cross transitions are encountered: For the (unusual) case of very small $n_{\max}$ and very high $s_{\max}$ and $c_{\max}$, the disk accesses to append target states to different queues would be mostly random, but in practice (with low $s_{\max}$ and I/O buffering) they are almost purely sequential. A theoretical worst-case analysis would thus be too pessimistic to be useful. We consequently abstain from such an analysis, too, and rely on the experimental evaluation of Section 5.

However, it is clear that the structure of the model w.r.t. the partitioning function will have a high impact on performance in general; in particular, a low number of cross edges is most desirable for the exploration algorithm presented here. Ideally, the partition graph is also forward-acyclic. In that case, two iterations of the outermost loop suffice: All states are explored in the first iteration, and the second only corrects the preliminary indices.

4 Disk-Based Partitioned Value Iteration

The result of the partitioned exploration presented in the previous section is a set of files in inverse-sequential format for the partitions of the state space. As mentioned in Section 1, value iteration can update the states in any order, as long as the maximum error for termination is computed in a way that takes all states into account. We can thus apply value iteration in a block-iterative

manner to the partitions of the state space as shown in Algorithm 3. The vector of values for each partition is stored in a separate file on disk. In lines 1 to 5, these files are created with the initial values based on whether a state is in the target set F. The actual value iterations are then performed in lines 6 to 20. For each partition, we need to load the sparse matrix-style representation of this part of the MDP into memory in the random-access format of Figure 1, plus the values for the current partition (line 9), and those of its successors (lines 10). The values of the successor partitions are needed to calculate the current state's new value in line 15 in presence of cross transitions. Memory usage is thus in $O(m_{\max} + s_{\max} \cdot n_{\max})$, where $m_{\max}$ is the maximum over all partitions of the sum of the number of states, transitions and branches. The I/O complexity is in $O(i \cdot p \cdot (\text{scan}(m_{\max}) + (s_{\max}+1) \cdot \text{scan}(n_{\max})))$ where i is the number of iterations of the outermost loop starting in line 6 and p is the total number of partitions.

In contrast to the exploration phase, the performance of this disk-based value iteration is not directly affected by the number of cross transitions. However, the number of successor partitions, i.e. $s_{\max}$, is crucial. An additional consideration is the way that values propagate through the partitions. The ideal case is again a forward-acyclic partition graph, for which a single iteration of the outermost loop (line 6) suffices since we iterate over the partitions in reverse order (line 8).

For expected rewards, we additionally need to precompute the sets of states that reach the target set with probability one and zero as mentioned in Section 2. The standard graph-based fixpoint algorithms used for this purpose [17] can be changed to work in a block-iterative manner in the same way as value iteration.

5 Evaluation

In this section, we investigate the behaviour of our disk-based probabilistic model checking approach and its implementation in mcsta on five models from the literature. Experiments were performed on an Intel Core i7-4650U system with 8 GB of memory and a 2 TB USB 3.0 magnetic hard disk, running 64-bit Windows 8.1 for mcsta and Ubuntu Linux 14.10 for PRISM version 4.2.1. We used a timeout of 12 hours. Memory measurements refer to peak working/resident sets. Since mcsta (implemented in C#) and parts of PRISM are garbage-collected, however, the reported memory usages may fluctuate and be higher than what is actually necessary to solve the task at hand. Our experiments show what the disk-based approach makes possible on standard workstation configurations today; by using compute servers with more memory, we can naturally scale to even larger models.

Detailed performance results are shown in Table 1. State space sizes are listed in *millions* of states, so the largest model has about 3.5 *billion* states. Columns "exp" and "chk" show the runtime of the exploration and analysis phases, respectively, in *minutes*. Columns "GB" list the peak memory usage over both phases in *gigabytes*. We show the performance of mcsta without using the disk to judge the overhead of partitioning and disk usage. Where possible, we also compare with PRISM, which does not use the disk, but provides a semi-symbolic HYBRID engine that uses BDD to compactly represent the states, transitions and branches while

keeping the entire value vector(s) in memory during value iteration (limiting its scalability), and a fully symbolic MTBDD engine that also uses BDD for the value vector. The HYBRID engine does not support expected rewards.

Compression. As all file accesses are sequential, we can use generic lossless compression to reduce disk accesses. Using the LZ4 algorithm [29], we achieved a 7× to 10× reduction in disk usage on our examples. We observed almost no change in runtime with compression enabled, so the extra CPU time is outweighed by reduced disk I/O. Compression thus lowers disk usage at no runtime costs.

Partitioning functions. The actual performance of our approach depends on the structure of the model and its interplay with the partitioning function. Scalability hinges on the function's ability to distribute the states such that the largest partition and the values of its successors fit into memory. The problem of automatically constructing a good partitioning function has largely been solved in prior work, and many techniques, like the ones described and referenced in [16], are available, but they are not yet implemented in mcsta. For our evaluation, we thus use relatively simple manually specified partitioning functions.

CSMA/CD: The MDP model of the IEEE 802.3 CSMA/CD protocol from the PRISM benchmark suite. It was manually constructed from a PTA model using the digital clocks approach. It has parameters N, the number of communicating nodes, and K, the maximum value of the backoff counter. The nodes count the number of collisions they encounter when trying to send a message. We partition according to the sum of the collision counters of the nodes. The resulting partition graph is forward-acyclic since these counters are only incremented, and $s_{\max} = N$. However, due to using the sum of several values for partitioning, the states are not evenly distributed over the partitions.

We first report on the performance of computing the minimum probability of any node eventually delivering its message with fewer than K collisions (model $\text{CSMA/CD}^{N,K}_{1\times\text{P}}$ in Table 1, with $1\times\text{P}$ indicating that one reachability probability is computed), and then on computing the max. and min. expected times until all nodes have delivered their message (model $\text{CSMA/CD}^{N,K}_{2\times\text{E}}$, where $2\times\text{E}$ indicates that we compute two expected-reward values). All MDP are only medium-sized. Our disk-based technique achieves performance comparable to the semi-symbolic approach here, which however does not support expected rewards. The fully symbolic approach has significantly higher runtimes for those properties.

Randomised Consensus: The PRISM benchmark of the randomised consensus protocol of N actors doing random walks bounded by K to reach a common decision. We partition according to the value of the shared counter variable. The resulting partition graph is strongly connected with $s_{\max} = 2$. We use $\epsilon = 0.02$ during value iteration (instead of the default $\epsilon = 10^{-6}$ as in the other examples). The MDP appear medium-sized in terms of states, but have about 5× as many transitions and 7× as many branches as states, so should be considered large.

Table 1. Evaluation results (millions of states, minutes, and gigabytes of memory)

	model		in-memory (mcsta)			**disk-based (mcsta -l)**					semi-symbolic (Prism hybrid)			fully symbolic (Prism mtbdd)		
	params	states	exp	chk	GB	p	$n_{\max}$	exp	chk	GB	exp	chk	GB	exp	chk	GB
CSMA/CD$^{N,K}_{1\times P}$	3, 4	1.5	0.1	0.0	0.3	12	0.4	0.2	0.0	0.2	0.0	0.2	0.2	0.0	0.5	0.3
	3, 5	12.1	1.1	0.1	2.6	15	2.6	1.3	0.1	0.7	0.1	1.6	0.5	0.1	4.0	0.4
	3, 6	84.9	> 8 GB			18	15.3	9.3	1.3	5.0	0.3	13.1	2.3	0.3	22.9	3.0
	4, 3	8.2	1.0	0.1	1.7	12	2.7	1.1	0.1	0.8	0.1	0.8	0.4	0.1	2.2	0.4
	4, 4	133.3	> 8 GB			16	33.0	19.1	2.2	6.6	0.4	17.6	3.6	0.6	21.7	5.1
	4, 5	2596.0						> 8 GB			> 8 GB			> 12 h		
CSMA/CD$^{N,K}_{2\times E}$	3, 4	1.5	0.1	0.1	0.3	12	0.4	0.2	0.2	0.2				0.0	18.1	0.4
	3, 5	12.1	1.1	1.5	2.6	15	2.6	1.3	1.7	0.7	n/a			0.1	96.9	4.7
	3, 6	84.9	> 8 GB			18	15.3	9.3	19.4	5.0				0.3	707.0	5.1
	4, 3	8.2	1.0	0.9	1.7	12	2.7	1.1	0.9	0.8				0.1	92.4	0.5
	4, 4	133.3	> 8 GB			16	33.0	19.1	16.5	6.6	n/a			0.5	637.3	5.5
	4, 5	2596.0						> 8 GB						> 12 h		
Consensus$^{N,K}_{2\times P}$	8, 2	61.0				5	16.8	10.5	104.9	6.4	0.0	28.3	1.6	0.0	5.4	0.3
	8, 3	87.9				7	16.8	16.0	200.6	4.3	0.0	65.1	2.3	0.0	10.1	0.4
	8, 4	114.8				8	16.8	21.8	347.5	7.3	0.0	121.4	2.9	0.0	17.5	0.4
	8, 5	141.6	> 8 GB			10	16.8	27.2	484.9	6.8	0.0	193.4	3.6	0.0	25.1	0.4
	8, 6	168.5				12	16.8	33.9	660.3	6.9	0.0	260.6	4.2	0.0	38.9	0.4
	8, 7	195.4						> 12 h			0.0	361.6	4.9	0.0	49.9	0.4
WLAN$^{K}_{1xP}$	1	718.0				203	11.5	177.3	8.5	3.0				715.3	4.3	5.8
	2	1197.9				337	12.0	283.5	15.7	3.0						
	3	1685.0	> 8 GB			471	13.1	392.2	23.4	3.0	> 8 GB			> 12 h		
	4	2186.7				605	15.1	502.6	30.7	3.5						
WLAN$^{K}_{1xE}$	1	718.0				203	11.5	177.3	52.4	3.0						
	2	1197.9				337	12.0	283.5	72.0	3.0						
	3	1685.0	> 8 GB			471	13.1	392.2	94.2	3.0	n/a			> 12 h		
	4	2186.7				605	15.1	502.6	114.0	3.5						
BRP$^{N,TD}_{6\times P}$	64, 16	18.7	1.5	0.2	3.8	65	0.3	1.8	0.5	0.2	23.0	56.8	1.0			
	128, 16	37.4	3.1	0.5	7.3	129	0.3	3.7	0.9	0.2	34.7	150.4	1.4	error		
	64, 32	70.7				65	1.2	7.4	1.8	0.5	89.4	345.2	2.4			
	128, 32	141.5	> 8 GB			129	1.2	15.3	3.4	0.5	> 12 h			error		
BRP$^{N,D}_{2\times TP}$	64, 256	355.7				577	1.5	40.7	3.3	0.6				122.6	38.2	2.6
	128, 256	715.6	> 8 GB			1153	1.5	93.0	60.6	0.6	> 8 GB			> 12 h		
	64, 512	1773.7				1089	4.8	203.1	18.8	1.6						
	128, 512	3573.3	> 8 GB			2177	4.8	418.5	38.1	1.8	> 8 GB			> 8 GB		
File server$^{C,D}_{2\times TP}$	5, 100	18.0	1.4	0.5	5.4	102	0.2	2.0	0.4	0.2						
	5, 200	41.2				202	0.2	4.7	1.0	0.2						
	5, 400	87.8	> 8 GB			402	0.2	10.5	2.1	0.2	n/a			n/a		
	5, 800	180.9				802	0.2	22.4	4.3	0.2						
	10, 100	34.0				102	0.4	4.0	0.9	0.2						
	10, 200	77.1				202	0.4	9.6	1.9	0.2						
	10, 400	163.4	> 8 GB			402	0.4	20.4	4.1	0.3	n/a			n/a		
	10, 800	335.9				802	0.4	43.9	8.6	0.3						
	params	states	exp	chk	GB	p	$n_{\max}$	exp	chk	GB	exp	chk	GB	exp	chk	GB

We check the two probabilistic reachability properties originally named "C_1" and "C_2". The fully symbolic technique completes exploration and analysis much faster than our disk-based approach. This is because this model is a benchmark for value iteration, with values propagating in very small increments back-and-forth through all the states and thus partitions. Still, we observe that $n_{\max}$ is invariant under K, so our technique will be able to check this model for $N = 8$ and any value of K without running out of memory—if given enough time.

Wireless LAN: The MODEST PTA model [21] of IEEE 802.11 WLAN, based on [28]. So far, this protocol has only been analysed with reduced timing parameters to contain state space explosion. We use the original values of the standard for a 2 Mbps transmission rate instead, including the max. transmission time of 15717 µs, with 1 µs as one model time unit. Parameter K is the maximum value of the backoff counter. We partition according to the first station's backoff counter, its control location, and its clock. The resulting partition graph has some cycles with $s_{\max} = 3$. Exploration needs 5 iterations of the outermost loop of Algorithm 2 in all cases. We compute the maximum probability that either station's backoff counter reaches K (model $\text{WLAN}^K_{1\times\text{P}}$ in Table 1) as well as the maximum expected time until one station delivers its packet ($\text{WLAN}^K_{1\times\text{E}}$).

BRP: The MODEST PTA model of the Bounded Retransmission Protocol (BRP) from [21]. Parameters are N, the number of data frames to be transmitted, MAX, the bound on the retries per frame, and TD, the maximum transmission delay. We fix $MAX = 12$. We partition by the number of the current data frame to analyse the model's six probabilistic reachability properties ($\text{BRP}^{N,TD}_{6\times\text{P}}$). This leads to the ideal case of a forward-acyclic partition graph with $s_{\max} = 1$. We also analyse two time-bounded reachability properties ($\text{BRP}^{N,D}_{2\times\text{TP}}$) with deadline D and fixed $TD = 32$, partitioning additionally according to the values of the added global clock. This leads to $s_{\max} = 2$. For the reachability probabilities, PRISM's MTBDD engine incorrectly reported probability zero in all cases. Our approach benefits hugely from having to perform far fewer total value iterations per state due to the favourable partitioning. In the reachability probabilities case, $n_{\max}$ is invariant under N, so we can scale N arbitrarily without running out of memory.

File Server: The STA file server model from [18]. C is the capacity of the request buffer. We compute the maximum and the minimum probability of a buffer overflow within time bound D. We cannot compare with PRISM because some features necessary to support STA cannot currently be translated into its input language from MODEST. Using our disk-based technique permits a finer abstraction for continuous probability distributions than before ($\rho = 0.01$ instead of 0.05). We partition according to the values of the global clock introduced to check the time bounds. This leads to the ideal case of an acyclic partition graph with $s_{\max} = 1$. The state space and number of partitions grow linearly in the time bound while $n_{\max}$ remains invariant. We can thus check time-bounded properties

for any large bound without exceeding the available memory, at a linear increase in runtime. This solves a major problem in STA model checking.

6 Conclusion

We have shown that the state space partitioning approach to using secondary storage for model checking combines well with analysis techniques built on graph fixpoint algorithms. We have used the example of MDP models and value iteration, but the same scheme is applicable to other techniques, too. In particular, the precomputation step for expected-reward properties is very close to what is needed for CTL model checking. Our technique is implemented in the mcsta tool of the MODEST TOOLSET, available at `www.modestchecker.net`. In our evaluation, we observed that it significantly extends the reach of probabilistic model checking. It appears complementary to the symbolic approach: On the model where our technique struggles, PRISM performs well, and where PRISM runs into memory or time limitations, our technique appears to work well. In particular, our approach appears to work better for expected-reward properties, and we have been able to defuse the crippling state space explosion caused by the deadlines of time-bounded reachability properties in PTA and STA models.

References

1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Commun. ACM 31(9), 1116–1127 (1988)
2. de Alfaro, L., Kwiatkowska, M.Z., Norman, G., Parker, D., Segala, R.: Symbolic model checking of probabilistic processes using MTBDDs and the Kronecker representation. In: TACAS. LNCS, vol. 1785, pp. 395–410. Springer (2000)
3. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
4. Baier, C., D'Argenio, P.R., Größer, M.: Partial order reduction for probabilistic branching time. Electr. Notes in Theoretical Comp. Science 153(2), 97–116 (2006)
5. Bao, T., Jones, M.D.: Time-efficient model checking with magnetic disk. In: TACAS. LNCS, vol. 3440, pp. 526–540. Springer (2005)
6. Barnat, J., Brim, L., Simecek, P.: I/O efficient accepting cycle detection. In: CAV. LNCS, vol. 4590, pp. 281–293. Springer (2007)
7. Bell, A., Haverkort, B.R.: Distributed disk-based algorithms for model checking very large Markov chains. Formal Methods in System Design 29(2), 177–196 (2006)
8. Bohnenkamp, H.C., D'Argenio, P.R., Hermanns, H., Katoen, J.: MoDeST: A compositional modeling formalism for hard and softly timed systems. IEEE Transactions on Software Engineering 32(10), 812–830 (2006)
9. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
10. Dai, P., Goldsmith, J.: Topological value iteration algorithm for Markov decision processes. In: IJCAI. pp. 1860–1865 (2007)
11. Deavours, D.D., Sanders, W.H.: An efficient disk-based tool for solving very large Markov models. In: Comp. Perf. Eval. LNCS, vol. 1245, pp. 58–71. Springer (1997)
12. Della Penna, G., Intrigila, B., Tronci, E., Zilli, M.V.: Exploiting transition locality in the disk based Murϕ verifier. In: FMCAD. LNCS, vol. 2517, pp. 202–219. Springer (2002)

13. Edelkamp, S., Jabbar, S.: Large-scale directed model checking LTL. In: SPIN. LNCS, vol. 3925, pp. 1–18. Springer (2006)
14. Edelkamp, S., Jabbar, S., Bonet, B.: External memory value iteration. In: ICAPS. pp. 128–135. AAAI (2007)
15. Edelkamp, S., Sanders, P., Simecek, P.: Semi-external LTL model checking. In: CAV. LNCS, vol. 5123, pp. 530–542. Springer (2008)
16. Evangelista, S., Kristensen, L.M.: Dynamic state space partitioning for external memory state space exploration. Sci. Comput. Program. 78(7), 778–795 (2013)
17. Forejt, V., Kwiatkowska, M.Z., Norman, G., Parker, D.: Automated verification techniques for probabilistic systems. In: SFM. LNCS, vol. 6659, pp. 53–113. Springer (2011)
18. Hahn, E.M., Hartmanns, A., Hermanns, H.: Reachability and reward checking for stochastic timed automata. ECEASST 70 (2014)
19. Hammer, M., Weber, M.: "To store or not to store" reloaded: Reclaiming memory on demand. In: FMICS/PDMC. LNCS, vol. 4346, pp. 51–66. Springer (2006)
20. Harrison, P.G., Knottenbelt, W.J.: Distributed disk-based solution techniques for large Markov models. In: Numerical Solution of Markov Chains. pp. 58–75 (1999)
21. Hartmanns, A., Hermanns, H.: A Modest approach to checking probabilistic timed automata. In: QEST. pp. 187–196. IEEE Computer Society (2009)
22. Hartmanns, A., Hermanns, H.: The Modest Toolset: An integrated environment for quantitative modelling and verification. In: TACAS. LNCS, vol. 8413, pp. 593–598. Springer (2014)
23. Hermanns, H., Wachter, B., Zhang, L.: Probabilistic CEGAR. In: CAV. LNCS, vol. 5123, pp. 162–175. Springer (2008)
24. Kwiatkowska, M.Z., Mehmood, R., Norman, G., Parker, D.: A symbolic out-of-core solution method for Markov models. Electr. Notes in Theoretical Comp. Science 68(4), 589–604 (2002)
25. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: CAV. LNCS, vol. 6806, pp. 585–591. Springer (2011)
26. Kwiatkowska, M.Z., Norman, G., Parker, D., Sproston, J.: Performance analysis of probabilistic timed automata using digital clocks. Formal Methods in System Design 29(1), 33–78 (2006)
27. Kwiatkowska, M.Z., Norman, G., Segala, R., Sproston, J.: Automatic verification of real-time systems with discrete probability distributions. Theoretical Computer Science 282(1), 101–150 (2002)
28. Kwiatkowska, M.Z., Norman, G., Sproston, J.: Probabilistic model checking of the IEEE 802.11 wireless local area network protocol. In: PAPM-PROBMIV. LNCS, vol. 2399, pp. 169–187. Springer (2002)
29. LZ4. `http://www.lz4.info/`, accessed: 2015-07-02
30. Mehmood, R.: Serial disk-based analysis of large stochastic models. In: Validation of Stochastic Systems. LNCS, vol. 2925, pp. 230–255. Springer (2004)
31. Norman, G., Parker, D., Sproston, J.: Model checking for probabilistic timed automata. Formal Methods in System Design 43(2), 164–190 (2013)
32. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley & Sons Inc., New York (1994)
33. Stern, U., Dill, D.L.: Using magnetic disk instead of main memory in the Murϕ verifier. In: CAV. LNCS, vol. 1427, pp. 172–183. Springer (1998)
34. Stewart, W.J.: Introduction to the numerical solution of Markov Chains. Princeton University Press (1994)
35. Timmer, M., Stoelinga, M., van de Pol, J.: Confluence reduction for probabilistic systems. In: TACAS. LNCS, vol. 6605, pp. 311–325. Springer (2011)